NPS52-82-005

NAVAL POSTGRADUATE SCHOOL

Monterey, California

Implementation of Parnas' it-ti Construct in LISP

A. Dain Samples

March 1982

Prepared for:

Chief of Naval Research
Arlington,Va 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

David A. Schrady
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-82-005	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Implementation of Parnas' it-ti Construct in LISP		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) A. Dain Samples		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01-10 N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE March 1982
		13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Chief of Naval Research Arlington,Va 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

language theory, programming constructs, LISP, structured programming.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

D. L. Parnas has recently proposed [1] a new programming control structure-the it-ti. This construct is a synthesis of several ideas in programming theory including iteration, conditionals, and Dijkstra's guards [2]. It has been implemented in a LISP interpreter [8] as a more structured replacement for the traditional prog construct. Several programming examples are given that compare the use of the it-ti with the more conventional programming constructs. These examples will also show that the it-ti fails to satisfy several criteria

for programming constructs including manageability and visibility. An appendix to this report contains an extension of Dijkstra's concept of the 'weakest precondition' to the it-ti.

Implementation of Parnas' it-ti Construct in LISP

A. Dain Samples

Abstract

D. L. Parnas has recently proposed [1] a new programming control structure - the it-ti. This construct is a synthesis of several ideas in programming theory including iteration, conditionals, and Dijkstra's guards [2]. It has been implemented in a LISP interpreter [8] as a more structured replacement for the traditional prog construct. Several programming examples are given that compare the use of the it-ti with the more conventional programming constructs. These examples will also show that the it-ti fails to satisfy several criteria for programming constructs including manageability and visibility. An appendix to this report contains an extension of Dijkstra's concept of the 'weakest precondition' to the it-ti.

1. Introduction

D. L. Parnas [1] has recently proposed a new programming control structure he calls the it-ti (iteration) construct after the fashion of do-od, if-fi, and case-esac. It is a synthesis of two concepts of structured programming (iteration and conditional execution) and is a direct descendant of Dijkstra's do-od and if-fi [2]. Reference [1] contains a detailed and mathematical treatment of the new construct for both non-deterministic and deterministic programming.

While the it-ti is certainly an interesting proposal and has filled a niche in Navlisp, we will see that it fails to satisfy several requirements of good programming.

To motivate the new construct, let us look at one of the thorns of structured programming [3]: the mid-exit loop, illustrated by a sequential array search. The task is to find an element K in an array A. If it is found, function F is passed the index of an element of A that contains K. If it is not found, function N is called with K as its parameter. A straightforward, non-structured implementation in pidgin-code* of this task might be:

* By pidgin-code we mean to indicate a language that incorporates features of several languages, and that we are free to modify as the need arises.

```
            index := 1;
loop:       if (index > sizeof(A)) then goto notfound;
            if (A[index] = K) then goto found;
            index := index + 1;
            goto loop;
notfound:   N(K);
            goto continue;
found:      F(index);
continue:
```

As Knuth [3] points out, there is really no satisfactory way to solve this problem with 'structured' programming. If we restrict ourselves and do not use any goto statements, we might solve the problem in one of the following ways:

Solution 1

```
found := false;
for i := 1 to sizeof(A) do
    if (A[i] = K) then
        found := true;
        index := i;
        endif;
    endfor;
if (found)
    then F(index);
    else N(K);
    endif;
```

Solution 2

```
found := false;
index := 1;
while (not found and index <= sizeof(A)) do
    if (A[index] = K) then found := true;
    else index := index + 1;
    endwhile;
if (found) then F(index);
    else N(K);
    endif;
```

Solution 3

```
index := 1;
while (index <= sizeof(A) cand A[index] <> K) do
    index := index + 1;
if (index > sizeof(A)) then N(K);
else F(index);
```

Solution 1 is obviously inefficient. The whole array is searched even if K is the first element. This could be an important consideration if the array A is large. Solution 2 is somewhat better, but the truly efficiency-minded point out that found

carries redundant information, is checked unnecessarily in the loop, and has to be checked again out of the loop. Solution 3 removes the variable but depends on conditional evaluation of boolean expressions. That is, the right-hand expression of a cand is evaluated only if the left-hand expression evaluates to true. In our example the cand prevents us from exceeding the array bounds of A. Even so, this solution still checks the value of index twice: once in the boolean expression of the while, and again upon exit from the while to determine why the loop terminated.

Note that the usual response to these criticisms is "Who cares?", and that is the correct response 99% of the time. The readability and maintainability of code must take precedence over efficiency almost all of the time. However, occasionally, a piece of code will occupy a critical position (e.g. in an inner loop) and the question becomes important. Again, the usual response is "Code it in assembly language". But this implies that readability and maintainability must be sacrificed in some situations for efficiency. A more desirable goal is a set of structured programming disciplines that promote readability and maintainability without sacrificing efficiency.

The Zahn-construct [3,4] attempted to solve this problem with an escape mechanism. The following implements the array-search with a Zahn construct (using the notation of Knuth [3]).

```
index := 1;
loop until found or notfound:
    if (A[index] = K) then found;
    index := index + 1;
    if (index > sizeof(A)) then notfound;
repeat;
then found     => F(index);
     notfound => N(K);
fi;
```

Some find this less satisfying than orthodox structured programming or programming with goto statements. Some of the problems with the Zahn-construct derive from the structure's lack of visibility and readability. For example, to trace the results of a particular event, (e.g. found) the reader must scan the then clause for the event label - the target of the event - much as he would have to scan for the target of a goto. While some of these problems might be resolved with appropriate syntactic 'sugar', the Zahn-construct is still a not-very-well disguised goto, replete with label. This can be appreciated better by drawing the control flow graphs for the if-then-else, repeat-until, etc., and comparing them with the control flow graph for the Zahn-construct (q.v. section 3, below): the Zahn-construct does not have a simple flow graph. About its only advantage is that the Zahn-construct forces the programmer to locally declare the labels in the context of use.

1. The It-ti Construct

Parnas has proposed a generalization of Dijkstra's do-od (iteration) and if-fi (conditional) constructs. The basic syntax of the it-ti is given below in a close facsimile of Parnas' notation* where <break/repeat> represents one of the keywords break or repeat.

```
it
      p1 -> s1 <break/repeat>
      p2 -> s2 <break/repeat>
      p3 -> s3 <break/repeat>
      :
      :
      pn -> sn <break/repeat>
ti
```

The semantics of the it-ti are straightforward. The predicates pi are evaluated sequentially until one evaluates to true. If pj is true, then the sequence of statements sj are evaluated. The last element of sj is a keyword that specifies whether the it-ti is to be executed again (repeat), or exited (break). It is an error if none of the predicates pi evaluate to true: in this case, the program aborts.

The following shows how an it-ti could be coded in a conventional programming language like Pascal, Algol, or FORTRAN.

```
ithead:
    if (p1) then begin
        s1;
        goto <label>; { where <label> is ithead (a repeat)  }
                      {                 or ittail (a break) }
        end
    if (p2) then begin
        s2;
        goto <label>;
        end
    :
    :
    if (pn) then begin
        sn;
        goto <label>;
        end
    error("it-ti fall through");
ittail:
    {rest of program}
```

---

* The notation in this report differs from Parnas in: (1) no elseor - the predicates are assumed to be executed deterministically and sequentially; (2) no up and down arrows - the keywords repeat and break are used instead.

The array search problem might be implemented using the it-ti as follows.

```
index := 1;
it
   index > sizeof(A) -> N(K); break;
   A[index] = K      -> F(index); break;
   true              -> index := index + 1; repeat;
ti
```

This seems to solve most, if not all, of the problems we have mentioned: there is no redundant boolean variable like 'found'; the search stops when the element is located; there are no redundant tests; there are no labels that redefine the 'shape' of the structure; the targets of the gotos implied by the repeats and breaks are visible and fixed; and any compiler worth its salt can factor out the constant boolean expression at code-generation time. Here we have a tight, efficient sequence of code.

The traditional control structures are available using the it-ti.

if-then-else:

```
if (b) then              |  it
    (body1)              |     b    -> body1; break;
else                     |     true -> body2; break;
    (body2)              |  ti
endif                    |
```

for-loop:

```
for i := 1 to n do       |  i := 1;
    (body)               |  it
    end;                 |     i <= n  -> (body) ;
                         |                i := i + 1;
                         |                repeat;
                         |     true    -> break;
                         |  it
```

while-loop:

```
while (b) do             |  it
    (body)               |     b    -> (body); repeat;
    end;                 |     true -> break;
                         |  ti
```

case-statement:

```
case x of                   it
    x1: body1;                  x = x1  -> body1; break;
    x2: body2;                  x = x2  -> body2; break;
     :                            :
    xn: bodyn;                  x = xn  -> bodyn; break;
    end;                    ti
```

repeat-loop:

```
repeat                      it
    (body)                      init or b -> (body); repeat;
    until b;                    true      -> break;
                            ti
```

3. Evaluation of the it-ti

There are several valid criticisms of the it-ti. For example, in order to handle the "execute at least once" loops (e.g. repeat-until, and the FORTRAN-66 DO-loop) a new feature must be added to the it-ti construct (see the repeat-loop implementation, above): associate with each it-ti an 'init' variable that is true only on the first iteration. And to prevent unnecessary evaluation of the boolean 'b' in this structure, we must also require conditional evaluation of boolean expressions (probably not an unreasonable requirement, but one forced on us by the structure of the it-ti, nevertheless).

Note also that the it-ti does not provide for efficient implementation of the case-statement. Without further enhancement, the it-ti deprives us of access to jump-table or hash-table implementation of selection constructs. The straightforward if-then-elseif approach to selection is not appropriate for all contexts.

Furthermore, practically speaking, the it-ti is not one control structure but a family of control structures. Unlike a conventional repeat or while in which we know the control flow entailed by the structure without knowing the details of the code, the introductory keyword of the it-ti does not give us any information about the control flow of the following code. It simply announces that a section of code has been reached which will contain alterations to sequential control flow. The reader will have to examine the body of each it-ti to understand the nature of the control flow. This criticism can be made concrete by observing that the 'traditional' control structures have a topologically constant flow chart representation as shown in Figures 1 - 5.

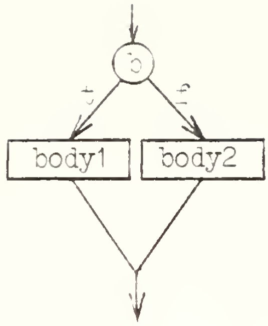

Figure 1. If-then-else

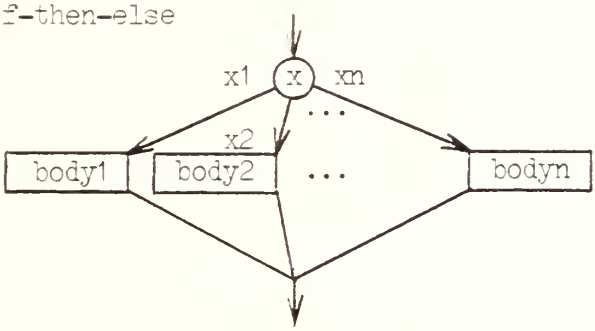

Figure 2. Case statement

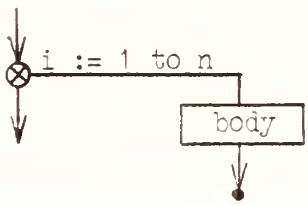

Figure 3. For-loop

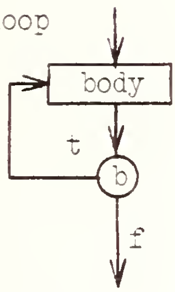

Figure 4. Repeat-loop

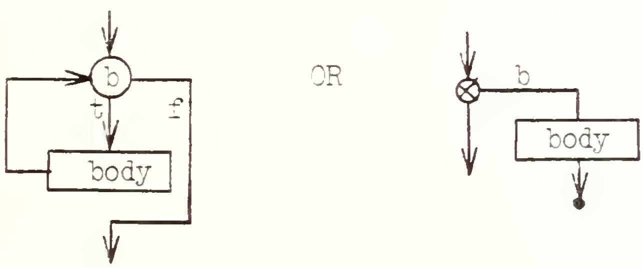

Figure 5. While-loop

No matter what 'b', 'x', or 'body' may be, the characteristics of the traditional control structures can be captured with a single

graph. Unfortunately, as for the Zahn-construct, there is no corresponding flow graph for the it-ti. About the best that can be done is to indicate a pseudo-decision node at the end of each guard clause:

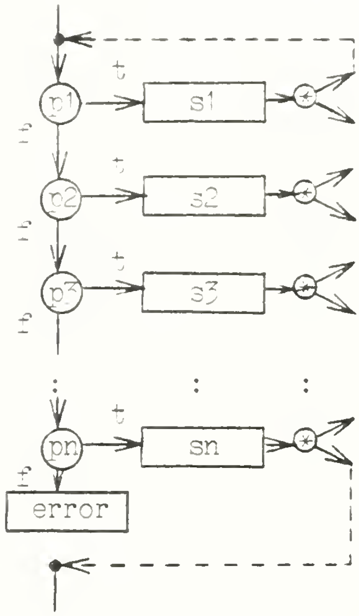

Figure 6. it-ti

They are not true decision nodes because there is no run-time decision to repeat or break: the programmer makes that decision at design time.

Since an it-ti with N guard clauses has 2**N possible flow-graphs, a large it-ti, can be quite complex and obscure, and fail to satisfy the need for readable, maintainable code. One would hope that this additional complexity would be justified by an improvement in our ability to express algorithms, especially 'simple' ones. But, there is still a simple control flow graph (from Knuth [3]) that the it-ti cannot handle without contortion. Dijkstra called it the loop that is executed "n and a half times" (see Figure 7).

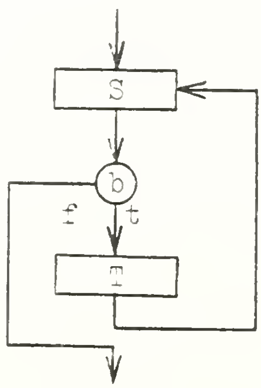

Figure 7. Dijkstra's n-and-a-half-times loop

If S is empty we have a while loop, and if T is empty we have a repeat loop. In spite of the fact that the it-ti allows us to

hand-craft a control structure to meet our needs, the "n and a half times loop" does not have a simple implementation using it. About the best we can do is to duplicate S, or introduce a boolean variable.

```
S;
it
   b    -> T; S; repeat;
   true -> break;
ti
```

```
toggle := true;
it
   toggle -> S; toggle := false; repeat;
   b      -> T; toggle := true;  repeat;
   true   -> break;
ti
```

This relatively simple control-flow graph can be implemented with an it-ti if we make a further extension to the construct. If we allow continue (fall through to the next guarded statement) as an alternative to break or repeat then

```
it
   true -> S; continue;
   b    -> T; repeat;
   true ->    break;
ti
```

would do precisely what we wanted. However, continue, like the init variable introduced earlier, is not clean and is very ad hoc in nature. It increases the complexity of the construct (there are now 3**N possible flow graphs) and does not enhance the visibility of the code.

And now, in order to implement intuitively simple programs, we find ourselves going through the same contortions with the it-ti that we have gone through with traditional programming structures.

4. LISP

Before we reject the it-ti entirely, however, let us look at one context in which the it-ti has proven effective.

Veteran LISP programmers may have a feeling of deja-vu with the it-ti. In very many ways, it resembles the cond conditional of LISP except that the cond does not iterate. For non-veteran LISP programmers, a short discussion of the control-flow structures of LISP follows.

4.1 LISP functions

In most programming languages, a function is called with syntax that looks something like

```
funcname (parm1, parm2, ...)
```

A LISP function application looks like:

```
(funcname parm1 parm2 ...)
```

4.2 cond

The syntax of LISP's control structure for conditional execution, the cond, is as follows.

```
(cond (p1 s1)
      (p2 s2)
         :
         :
      (pn sn)
      )
```

The pi are predicate expressions evaluating to the LISP equivalents of true or false. The pi are executed sequentially until one of them, say pj, evaluates to true. At that point, the list of statements sj is evaluated. When all of the statements in statement list sj are evaluated, interpretation commences at the first program statement after the final parenthesis of the cond.

4.3 prog

McCarthy [7] reports that he originally intended that LISP would be a FORTRAN-like list processing language with the addition of recursion and conditional evaluation of boolean expressions. He also admits that prog looks like an "afterthought", and that its design was an "afterthought". But it is this control structure that provides iteration via labels and go. If we assume that (A n) is a function that returns the nth element of the array A, then the array search problem could be written in LISP as follows.

```
(prog (index)
    (setq index 1)
    loop    ;this is just a label, not a LISP keyword
        (cond ((greaterp index (sizeof A)) (go notfound))
              ((equal (A index) K) (go found))
              )
        (setq index (add1 index))
        (go loop)
    notfound
        (N K)
        (return)
    found
        (F index)
        (return)
    )
```

A more efficient LISP program to perform this search would be:

```
(prog (index)
    (setq index 1)
    loop
        (cond ((greaterp index (sizeof A)) (N K) return)
              ((equal (A index) K)         (F index) return)
              )
        (setq index (add1 index))
        (go loop)
    )
```

One can see that this transliteration from our pidgin-code into LISP results in FORTRAN-like code which, when one thinks about it, is a far cry from the programming style indicated by the rest of the LISP language: this goto style of programming is an abdication of the applicative nature of LISP.

5. An alternative to prog

A moment's thought will convince you that it is easy enough to incorporate the it-ti construct into LISP. The syntax of the cond is modified to accept one of the keywords break or repeat at the end of the statement list. The keyword break specifies that control exits the cond. The keyword repeat specifies that the cond is to be repeated. I.e.,

```
(cond (p1 s1 [repeat/break])
      (p2 s2 [repeat/break])
         :
         :
      (pn sn [repeat/break])
      )
```

There are some differences between the enhanced cond in Navlisp and Parnas' strict definition of the it-ti. For upward

compatibility with current implementations of LISP. The repeat/break is optional since the conventional cond is defined as if break were at the end of each statement list. And for the same reason, the cond does not abort if all of the guards/predicates are false. Unfortunately, this results in the loss of one of the distinct advantages of the it-ti: providing an effective run time check that each case has been anticipated.

With this enhancement to cond, the coding of our example is concise:

```
(setq index 1)
(cond
  ((greaterp index (sizeof A)) (N K)                         break)
  ((equal (A index) K)         (F index)                     break)
  (T                           (setq index (add1 index)) repeat)
  )
```

Note again that the keyword break is not necessary since that is the default action of the LISP cond.

5.1 Examples

Winston [5], page 328, defines a non-recursive factorial function which is reproduced below:

```
(defun factorial (n)              ; written in Maclisp
       (prog (result counter)
             (setq result 1)
             (setq counter n)
             loop                 ; note the label
             (cond ((zerop counter) (return result)))
             (setq result (times counter result))
             (setq counter (sub1 counter))
             (go loop)))
```

The following shows the result of defining the function using the modified cond.

```
(defun factorial (n)              ; written in Navlisp
       (let ((result 1) (counter n))
            (cond ((zerop counter) result break)
                  (T (setq result (times counter result))
                     (setq counter (sub1 counter))
                     repeat))))
```

The let used in the above example is a LISP function which, like the begin of Algol and the curly brace '{' of C, defines an environment with local variables. That is, let is simply a prog without the complications of labels, go. and return. In Maclisp, let, prog, and an unmodified cond are available. In Navlisp, only the let and the enhanced cond are necessary.

McCarthy [6], page 25, illustrates the use of prog with a program to compute the length of a list. His version of this function uses prog:

```
(defun length (list)
    (prog (u v)
          (setq v 0)
          (setq u list)
        a (cond ((null u) (return v)))
          (setq u (cdr u))
          (setq v (add1 v))
          (go a) ))
```

The following is a version using the modified cond.

```
(defun length (list)
    (let ((v 0) (u list))
         (cond ((null u) v break)
               (T (setq u (cdr u))
                  (setq v (add1 v))
                  repeat))))
```

A more intricate example comes from Winston [5], page 334. If we assume that (matrix i j) is a function that returns the (i,j) element from a two-dimensional array named matrix, then Winton's version using prog is:

```
(defun print-matrix (n m)       ; written in Maclisp
    (prog (i j)
          (setq i 0)
          row-loop
          (cond ((equal i n) (return nil)))
          (terpri)                ; prints an end-of-line
          (setq j 0)
          column-loop
          (cond ((equal j m) (go next-row)))
          (princ (matrix i j))
          (princ '| |)            ; prints a blank
          (setq j (add1 j))
          (go column-loop)
          next-row
          (setq i (add1 i))
          (go row-loop)))
```

This program prints out the elements of a two-dimensional array a row at a time. The following is the Navlisp equivalent using the it-ti form of cond:

```
(defun print-matrix (n m)        ; written in Navlisp
      (let ((i 0) (j))   ; j is not initialized
         (cond ((equal i n) nil break)
               (T (terpri)
                  (setq j 0)
                  (cond ((equal j m) break)
                        (T (printf (matrix i j) " ")
                           (setq j (add1 j))
                           repeat))
                  (setq i (add1 i))
                  repeat))))
```

Our final example is the LISP interpreter function evcon coded in Navlisp.

```
(defun evcon (beg a)
       (let ((c beg))
       (cond ((null c) break)
             ((eval (caar c) a)
                 (let ((slist (cdar c)) (s (car (cdar c))))
                 (cond ((null s) (set 'c (cdr c)) break)
                       ((atom s)
                           (cond ((equal s 'repeat) (set 'c beg)
                                     break)
                                 ((equal s 'break ) (set 'c nil)
                                     break)
                                 (T (eval s a)
                                     (set 'c (cdr c))
                                     break))
                           break)
                       (T  (eval s a)
                           (set 'slist (cdr slist))
                           (set 's (cond ((not (null slist))
                                                (car slist))))
                           repeat)))
                 repeat)
             (T (set 'c (cdr c)) repeat)
       )     ))
```

6. Conclusions

Computer Science as a field is always on the lookout for programming concepts that simplify the art of programming. Parnas' it-ti is an interesting proposal to that end, but does have several serious drawbacks. However, the it-ti has proved itself a very apt programming structure for LISP, allowing the complex prog to be replaced by an enhancement to cond. The new cond is slightly more complicated than the old, but allows the writing of iterative programs whose structures are more within the programming spirit of LISP. This structure has been implemented in Navlisp and has proved itself to be both compact* and sufficient.

Appendix

In [2], Dijkstra develops a programming language conducive to correctness proofs in which he defines two constructs he calls if-fi and do-od. He introduces the concept of the 'weakest precondition', written wp(S,R) and paraphrased as 'The necessary conditions which guarantee that execution of statement S will leave the computation in state R.' Parnas [1] cites Dijkstra [2] as his inspiration for the it-ti, noting that it is a modification of the do-od and the if-fi. He does not, however, attempt to apply Dijkstra's 'weakest precondition' to the it-ti. For completeness, it is included here. The reader is referred to [2] for a complete discussion of the development of the weakest precondition for the do-od and if-fi. It should also be noted that the following discussion assumes non-determinacy, as do Dijkstra and Parnas.

Let IT refer to the it-ti:

```
it
   B1 -> S1
   B2 -> S2
      :
      :
   Bn -> Sn
ti
```

where the last statement in each Si is either break or repeat. At least one of the Bi will evaluate to true (otherwise the results are undefined and will abort execution of the program). That is, using the notation in [2]**:

```
(Ei : 1 <= i <= n : Bi)
```

Since the construct is non-deterministic, we can re-arrange the statements such that:

```
(Em : (0 <= m <= n) :
      (Ai : [(1 <= i <= m) and (Si ends with 'repeat')]
         or [(m <= i <= n) and (Si ends with 'break') ] ))
```

If m = 0 then the construct corresponds to Dijkstra's if-fi, and if m = n the construct is an infinite loop. This is effectively equivalent to the following:

---

* It may interest some to know that it required only six additional lines of C-code to the Navlisp interpreter to implement the enhanced cond.

** The notation uses Ei to mean 'there exists an i', and Ai to mean 'for all i'.

```
do
   B1 -> S1
   B2 -> S2
      :
   Bm -> Sm
od
if
   B(m+1) -> S(m+1)
      :
   Bn     -> Sn
   true   -> error
fi
```

(The last statement of the if is necessary if m = n.) Note that this assumes that for any iteration either

```
(Ai : Bi : (Aj : Bj : [i in (1,m)   and j in (1,m)]
                   or [i in (m+1,n) and j in (m+1,n)] ))
```

or that do-od/repeat statements take precedence over if-fi/break statements if more than one Bi is true. With these qualifications on the it-ti we can arrive at its weakest precondition. Let DO' represent that sub-part of the it-ti that repeats, and IF' represent that sub-part of the it-ti that breaks, then

```
wp(IT,R) = wp(DO', wp(IF',R) )
```

where (from [2])

```
wp(IF',R) = (Ej : m < j <= n : Bj) and
            (Aj : m < j <= n : Bj => wp(Sj,R))

wp(DO',R) = (Ek : k >= 0 : Hk(R))

Hk(R) = wp(IF, H(k-1)(R)) or H(k-1)(R)

H0(R) = R and not (Ej : 1 <= j <= m : Bj)
```

References

[1] Parnas, D. L.; An Alternative Control Structure and Its Formal Definition, paper draft.

[2] Dijkstra, Edsgar; A Discipline of Programming; Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

[3] Knuth, D. E.; "Structured Programming with goto Statements"; Computing Surveys, Vol. 6, No. 4, Dec. 1974.

[4] Zahn, Charles T., "A control statement for natural top-down structured programming", presented at the Symposium on Programming Languages, Paris, 1974.

[5] Winston, P. H., and Horn, B. K. P.; LISP; Addison-Wesley Publishing Company, Reading, Mass., 1981

[6] McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., Levin, M. I.; LISP 1.5 Programmer's Manual, M.I.T. Press; 2nd ed., 1965.

[7] McCarthy, J.; ACM SIGPLAN Proceedings of the History of Programming Languages Conference; June, 1978.

[8] Samples, A. D.; Navlisp Reference Manual, Computer Science Laboratory Technical Report NPS52-82-004, Naval Postgraduate School, 1982.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox LIbrary Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
A. Dain Samples, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
Professor Douglas R. Smith, Code 52Sc Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
Professor David Parnas 12503 Davan Drive Silver Spring, MD 20904	1
Chief of Naval Research Arlington, Va 22217	1